Guaranteeing the Metadata Update Atomicity in EXT4 File system[*]

Seongbae Son Jinsoo Yoo Youjip Won
Hanyang University, Seoul, Korea
{afireguy|jedisty|yjwon}@hanyang.ac.kr

ABSTRACT

In this paper, we address the issue of guaranteeing the atomicity of metadata update in a `write()` system call in EXT4 filesystem. Recent versions of EXT4 delay inserting the updated inode to the running journal transaction until the associated dirty pages are actually written to the disk. This is to avoid excessive `fsync()` overhead. While this approach effectively reduces the tail latency of `fsync()`, we found that it can incorrectly recover the file and it can expose the interim state of the inode to the application when the filesystem crashes unexpectedly. To address this problem, we propose *Delayed Inode Update*, DIU. Instead of separating the update of an inode and its insertion to the running transaction, we propose delaying the update until the associated inode is inserted into journal transaction. Delayed Inode Update is crafted not to entail any performance overhead nor does it increase the `fsync()` latency. With Delayed Inode Update, the average and the worst case latency of an `fsync()` decrease by 15% and 43% in a designated workload, respectively.

1 INTRODUCTION

EXT4 [28] filesystem is the most widely used filesystem in all range of computing platforms from the wearable device [1] to the enterprise server. EXT4 adopts block granularity and redo only physical logging [17] for crash recovery. While EXT4 journaling is known for its maturity and sophistication, it is still considered as one of the most significant impediments in the modern IO subsystem in fully exploiting hardware performance of the underlying storage [14].

[*]The initial version of the source code is publicly available at https://www.spinics.net/lists/linux-ext4/msg55460.html

APSys '17, September 2, 2017, Mumbai, India
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5197-3/17/09…$15.00
https://doi.org/10.1145/3124680.3124722

EXT4 filesystem maintains one active transaction in memory where the filesystem maintains the redo logs. This is a called running transaction. To avoid frequent journal commit [19], EXT4 filesystem aggregates updated blocks to the active transaction and flushes them either periodically or via an explicit call, e.g. `fsync()`. By aggregating multiple updated blocks, EXT4 can commit the journal transaction less frequently and the overhead of the filesystem journaling can be mitigated. This aggregated approach introduces another problem. The ordered mode, one of the journaling modes in EXT4, ensures that the dirty page cache entries are made durable before the associated metadata do. This is to avoid any inconsistency between the file blocks and the associated metadata, e.g. the block pointers point to the file blocks with garbage data. When an inode in the running transaction happens to have a large amount of dirty pages, e.g. downloading a file, committing a running transaction can take a prohibitive amount of time reaching as much as a few seconds [7]. When an application calls an `fsync()`, it may suffer from unexpectedly long latency due to the other inodes which are in the same running transaction.

By delaying the time when the updated inode is inserted to the running transaction, one can improve the worst case latency of `fsync()`. This is because `fsync()` only writes the metadata of the associated file to the journal area.

However, we have found that this approach can incorrectly recover the filesystem and it can leave one or more files in the inconsistent state in the case of unexpected system failure in the ordered mode journaling. We call this problem atomic metadata update failure.

In this work, we propose *Delayed Inode Update* (DIU) to guarantee the atomic update of the file size associated with a `write()` in EXT4. The delayed inode update technique postpones not only the insertion of the updated metadata into a journal transaction but also the update of the inode itself without compromising the atomicity of the inode update. DIU consists of two technical ingredients: *delayed filesize update* (DFU) and *delayed timestamp journaling* (DTJ). *Delayed filesize update* (DFU) guarantees the atomicity of metadata update operation. *Delayed timestamp journaling* (DTJ) effectively reduces the amount of metadata committed to the storage.

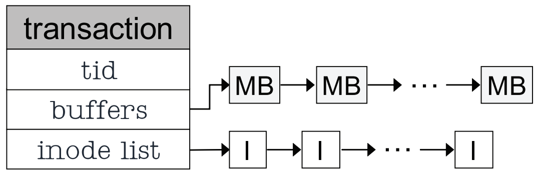


Figure 1: `transaction` structure of EXT4 journaling (Ordered mode). MB: metadata block, I: VFS inode

In the `fsync()` performance evaluation, DIU shows up to 15% and 43% lower latency than EXT4 in the average and the worst case latency.

The rest of the paper is organized as follows. Section 2 describes the basics of the EXT4 journaling and the write implementation. Section 3 describes the consistency guarantee issue in EXT4. Section 4 describes the design and the implementation of DIU. Section 5 shows the performance of DIU compared with EXT4. Section 6 describes the related work and Section 7 concludes the paper.

2 BACKGROUND

2.1 Journaling in EXT4

EXT4 filesystem provides three journaling modes: the write-back mode, the ordered mode, and the data mode [22]. The default is the ordered mode. In the ordered mode, the filesystem journals only the updated metadata. The filesystem guarantees that the dirty pages associated with the metadata being journaled are made durable before the journal transaction which contains the updated metadata is made durable [23].

To mitigate the overhead of filesystem journaling, EXT4 filesystem aggregates the blocks which are to be journaled and writes them as a single unit, called a journal transaction or a transaction for short.

A journal transaction consists of the transaction header block, one or more log blocks and the transaction commit block. A transaction is in one of three states; running, committing, and checkpoint. A running transaction is one where the applications or JBD2 thread are inserting the log records. The transaction is said to be in committing state if it is being written to the storage. The committing transaction is frozen. It stops accepting more log blocks. When an application tries to journal an updated block, a new running transaction is created if it does not exist. EXT4 has at most one running transaction and at most one committing transaction [28]. Once a transaction is committed, the state of a transaction changes to "checkpoint". There can be one or more transactions in the checkpoint state.

Fig. 1 shows the `transaction` structure of EXT4. Each `transaction` is assigned a unique identifier, `tid`. A `transaction` maintains a set of updated blocks, called `buffers` to be journaled. The set of buffers is maintained as a linked list of the pointers to the associated page cache entries. A `transaction` structure maintains a list of inodes, `inode list` associated with the log blocks.

EXT4 filesystem dedicates a separate thread for journaling, the Journaling Block Device 2(JBD2) daemon [16]. The JBD2 daemon runs periodically (default 5 seconds) [25] or is woken up by the `fsync()` system call [2].

Linux OS maintains two versions of the inode: VFS inode and EXT4 inode. EXT4 inode is the on disk representation of a file and is the filesystem specific data structure. VFS inode is intended to insulate the OS from the implementation details of the individual filesystems. It is a common representation of a file. When a file is opened, VFS inode is created and is initialized with the value associated with a given EXT4 inode. Usually, VFS inode is a superset of filesystem specific inode, e.g. EXT4 inode. Inode list in a journal transaction is a list of VFS inodes. The `inode list` is used to identify the dirty pages for a given files. When the JBD2 starts committing a journal *transaction* in ordered mode, it first scans the `inode list` and dispatches the write requests for the associated dirty pages. Once this is done, the JBD2 writes the transaction header block and the blocks in the `buffers` list to the journal region. The JBD2 daemon waits until the dirty page cache entries, the journal header block and the log blocks are made durable. Then, it issues the write request for journal commit block. Later, the committed metadata blocks are checkpointed to the original location in the filesystem partition by the `kworker thread` [7].

2.2 Types of `write()` in EXT4

We categorize the `write()` operations into four categories. First, we categorize `write` into two major types depending upon whether it allocates a new file block or not: an allocating write (AW) and a non-allocating write (NAW).

An allocating write is divided into two categories based upon whether the starting position is aligned with block boundary or not: aligned write (AW-A) and unaligned write (AW-U). Non-allocating write is further divided into two based upon whether the file size changes or not: size preserving (NAW-SP) and size-changing (NAW-SC). Fig. 2 illustrates an example for each of the four categories.

Each of write follows a different execution path in updating the associated metadata and in journaling them. Fig. 3 shows the individual sequences. EXT4 filesystem maintains two variables to represent a file size. One is in the VFS inode and the other is in the EXT4 inode. The metadata is updated as follows.

First, it updates the access time (atime) and modification time (mtime) of the VFS-inode and the EXT4-inode respectively. After the metadata block is inserted to the `buffers`

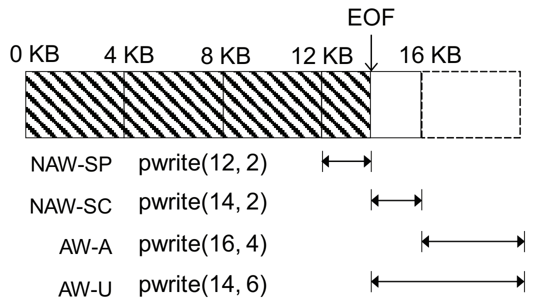


Figure 2: Types of `write()`: size preserving non-allocating write (NAW-SP), size changing non-allocating (NAW-SC), aligned allocating write (AW-A), and unaligned allocating write (AW-U)

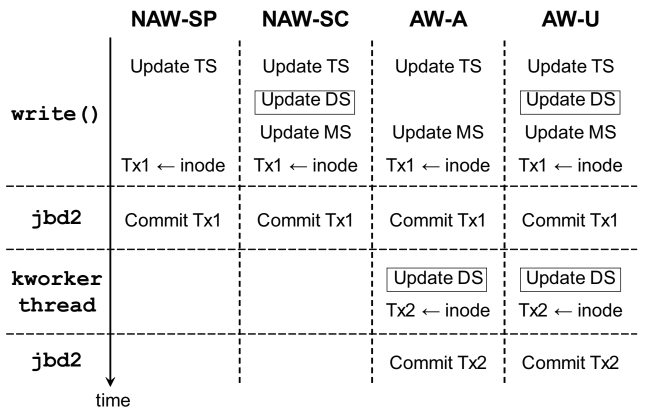


Figure 3: Metadata update and journaling for each write() type, TS: `atime` and `mtime`, DS: file size in EXT4 on-disk inode, MS: file size in VFS inode

list in the running transaction, it updates the page cache entries. Then, the file size at the EXT4 inode is updated. There is an important distinction in the way in which the file size is updated. In the size-changing non-allocating write, the file size of the EXT4 inode is increased by the size of appended data. In the unaligned allocating write, the file size of the EXT4 inode is updated to the sum of the already allocated blocks. In the size preserving non-allocating write and in the aligned allocating write, the file size of the EXT4 inode is not updated.

When the file size of the EXT4 inode is updated, the inode is inserted to the `inode list` of the running transaction. The implementation of the updating `inode list` depends on the kernel version. In the Linux kernel prior to version 3.8, `write()` system call inserts the updated inode to `inode list`. In the Linux kernel version 3.8 and later, the `write()` system call omits to insert the updated inode to the `inode list`. Fig. 4 illustrates the difference.

	NAW-SP	NAW-SC	AW-A	AW-U
`write()`		O		O
`fsync()`/kworker			O	O

Table 1: The number of updates of the file size in EXT4 inode

After the `inode list` is updated, the updated EXT4 inode is inserted to the `buffers` list of a running transaction. If the EXT4 inode is already inserted into the `buffers` list at the time of the timestamp [29] update, the insertion is skipped. Finally, the file size of the VFS inode is updated to its final value after the inode block is inserted to the `buffers` list. Fig. 3 illustrates how the individual fields at the metadata and the running transaction are updated.

In the size preserving non-allocating write, the file size of the EXT4 inode is not updated. In the size changing non-allocating write, the file size of the EXT4 inode is updated once, and the updated metadata block is inserted into the journal transaction once. In the aligned allocating write, the file size of the EXT4 inode is updated once, not by the `write()` system call but by the kworker thread. The updated metadata block is inserted to the journal transaction twice: one in the `write()` system call and the other by kworker or `fsync()`. In the unaligned allocating write, the file size of the EXT4 inode is updated twice: at the `write()` and at the kworker thread. The updated metadata block is inserted into journal transaction twice. Table 1 shows the number of file size updates of the EXT4 inode according to the write types.

3 PROBLEM STATEMENT

Delayed Inode Insertion is proposed to reduce the worst case latency of `fsync()` [27]. It fails to guarantee the atomicity of the metadata update operation[18] in EXT4. Due to Delayed Inode Insertion, the ordering constraint between the data block and the associated metadata in the ordered mode journaling can be compromised. There is a time interval during which the content of the `inode list` and the content of the `buffers` list of a running transaction do not coincide. The problem arises when the application calls an `fsync()` system call to persist the dirty pages and the metadata for a file. When the JBD2 daemon commits a running transaction, it is possible that some inodes in the running transaction are not present in the `inode list`. The JBD2 daemon fails to persist the dirty pages associated with these missing inodes when the inodes which are not present in the `inode list` are made durable through the journal commit. When the system crashes, the recovery module can recover the inodes in the journal region, but without the appropriate data blocks on disk.

The root cause of this problem is the metadata inconsistency in a journal transaction. Some of the updated inodes

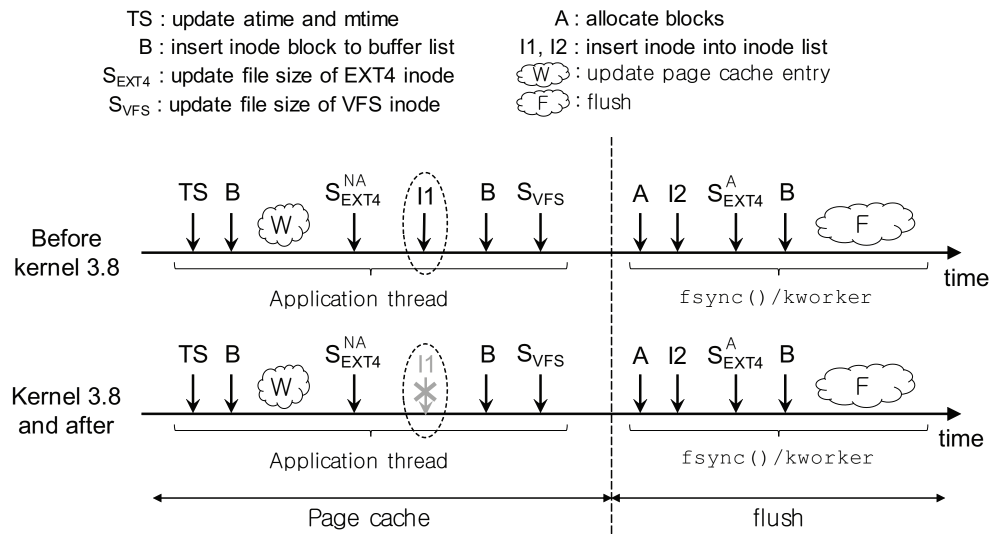

TS : update atime and mtime
B : insert inode block to buffer list
S_EXT4 : update file size of EXT4 inode
S_VFS : update file size of VFS inode
A : allocate blocks
I1, I2 : insert inode into inode list
W : update page cache entry
F : flush
Before kernel 3.8
TS B W S^NA_EXT4 I1 B S_VFS A I2 S^A_EXT4 B F
Application thread
fsync()/kworker
time
Kernel 3.8 and after
TS B W S^NA_EXT4 I1 B S_VFS A I2 S^A_EXT4 B F
Application thread
fsync()/kworker
time
Page cache
flush

Figure 4: Updating the metadata and the journal transaction. S^{NA}_{EXT4}: updating the file size in EXT4 inode in non-allocating write, S^{A}_{EXT4}: updating the file size in EXT4 inode in allocating write, S_{VFS}: updating the file size in VFS inode

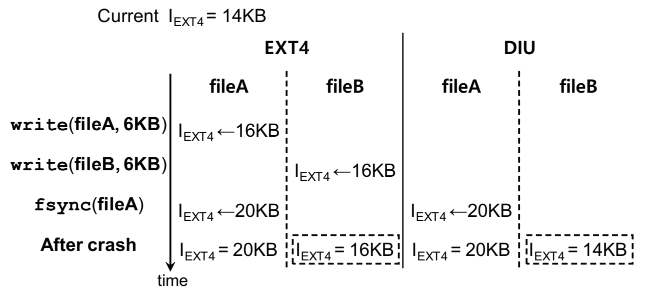


Figure 5: Example of AW-U (Unaligned allocating write) in EXT4 and DIU. I_{EXT4} : EXT4 inode

which have been inserted in the `buffers` list may not be present in the `inode list`. This happens due to the delayed inode insertion in the recent versions of EXT4. In Linux prior to version 3.8, an updated inode is inserted to the `buffers` list as well as `inode list` at the same time. It is guaranteed that the `buffers` list and `inode list` in a journal transaction are aligned with each other (Fig. 4).

In delayed inode insertion, when the application updates the size of a file, the associated inode may not be inserted to the `inode list` (I1 in Fig. 4) while the associated metadata block is inserted to the `buffers` list in the running transaction. This happens when the starting offset of a write operation is not aligned with the start of a file block; the size changing non allocating write(NAW-SC) and the unaligned allocating write(AW-U).

Fig. 5 shows an example. There are the two files, namely, fileA and fileB. The sizes of two files are 14 KB. An application appends 6 KB data blocks to fileA and fileB, respectively. Both are unaligned allocating write. In the context of the caller, the sizes of the two files are updated to 16 KB (I1 in Fig. 4). The metadata blocks associated with the updated inodes are inserted into the `buffers` list of a running transaction. When the application calls `fsync('fileA')`, it writes the data of fileA to the writeback cache in a storage and changes the file size to 20 KB. After the JBD2 daemon wakes up, the inode of fileB as well as the inode of fileA in the `buffers` list are committed to the storage. If a system crash occurs after the `fsync()` is completed, EXT4 recovers the inodes of fileA and fileB. The recovered inode of fileB has the updated file size, 16 KB, even though the data of fileB has not been made durable.

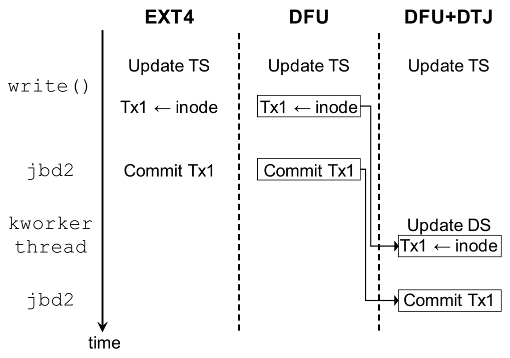
We have tested four other open source filesystems, XFS[26], F2FS[11], BTRFS[20], and ZFS[21] if they guarantee the atomicity in metadata update[1]. XFS, F2FS, and BTRFS passed the test, but ZFS has failed.

4 DELAYED INODE UPDATE

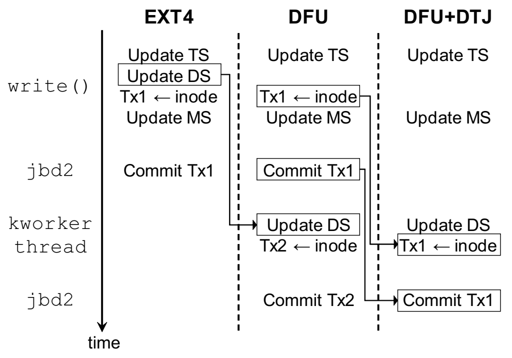
4.1 Delayed File Size Update

We postpone updating the file size until when the associated dirty pages are flushed to the storage. This is called the *Delayed Filesize Update* (DFU). Unlike EXT4, DFU postpones updating the file size to the point when the dirty pages

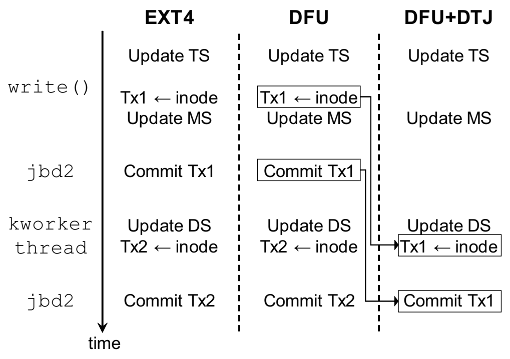
[1]the test code is available at `https://github.com/seongbaeSon/DIU.git`



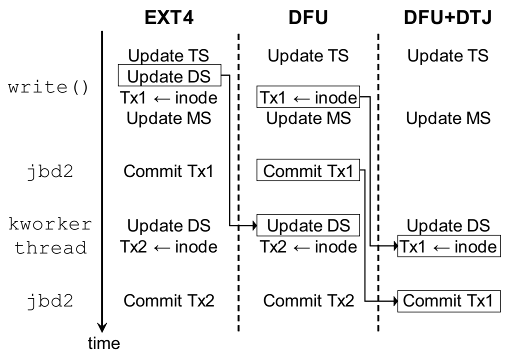
(a) size preserving non-allocating write



(b) size changing non-allocating write



(c) Aligned Allocating write



(d) Unaligned Allocating write

Figure 6: `write()` and filesystem journaling in EXT4, DFU, and DFU+DTJ. TS: `atime` and `mtime`, DS: file size of EXT4 on-disk inode, MS: file size of VFS inode

are flushed. In DFU, the filesystem checks if the file size in the EXT4 inode is same as that of the VFS inode when the kworker or `fsync()` flushes the dirty pages of the inode to the storage. If they do not match, the file size of the EXT4 inode is updated to the file size of the VFS inode and the updated EXT4 inode is inserted to the running transaction.

Fig. 6 illustrates how the metadata is updated in original EXT4 and in DIU, respectively. In the size preserving non-allocating write and the aligned allocating write, DFU does not bring any change (Fig. 6(a), Fig. 6(c)). In the size changing non-allocating write (NAW-SC) and in the unaligned allocating write (AW-U), the file size is updated in `write()` system call. In delayed inode update, we update the file size when the kworker thread flushes the dirty pages (Fig. 6(b) and Fig. 6(d)). Let us provide an example (DIU in Fig. 5). DFU does not change the file size within a `write()` system call. In DFU, the inode of fileA and the inode of fileB are not inserted into the `buffers` list of the running transaction. The file size is changed from 14 KB to 20 KB when the associated dirty pages are flushed to disk.

When an `fsync('fileA')` is called, the metadata block associated with the inode of fileA is inserted into the `buffers` list. In delayed inode update, the inode of fileB is still not inserted in the running transaction. The JBD2 daemon commits the journal transaction. If a system crash occurs after `fsync()` is completed, the inode of fileB is still 14 KB since the inode of fileB is not written in the journal area. DFU ensures the consistency of fileB by preventing the inode of fileB from being written in the journal area before the user data of fileB is flushed.

4.2 Delayed Timestamp Journaling

In EXT4, the updated inode is inserted to the running transaction twice: after updating the timestamp and after the block is actually allocated. We develop Delayed Timestamp Journaling to reduce the amount of IO associated with journaling and to reduce the `fsync()` latency.

In the allocating write, the inode is inserted into the running transaction twice. `write()` system call inserts the inode to a running transaction after it updates the timestamp(mtime and atime). The kworker thread or `fsync()` allocates the block, updates the file size and inserts the updated inode to the running transaction. It is possible that the JBD2 daemon is triggered between the two insertions. Then, the metadata block which contains the inode is written to the storage twice.

In Delayed Timestamp Journaling, the filesystem omits the first insertion of the two. It does not insert the inode to the running transaction when it updates the timestamp. 'DFU+DTJ' in the Fig. 6 illustrates this scheme.

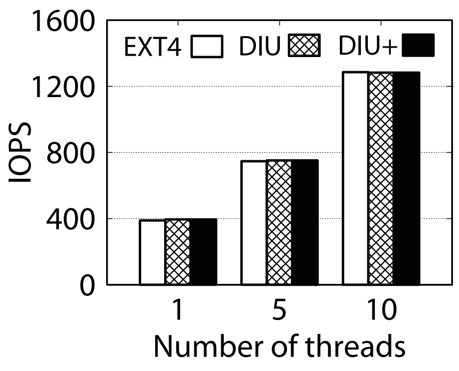
In the allocating write, Delayed Timestamp Journaling reduces the amount of IO associated with journaling which reduces the latency of `fsync()`. Suppose that we append 4 KB blocks to fileA and fileB respectively, and we call `fsync('fileA')`. In EXT4, the metadata block associated with fileA and the metadata block associated with fileB are inserted to the running transaction when the `write()` system call is called. The running transaction contains the metadata blocks associated with fileA's inode and the fileB's inode. If we use both Delayed File size Update (DFU) and Delayed Timestamp Journaling (DTJ), the running transaction contains only the metadata block associated with fileA. The running transaction becomes smaller when the `fsync('fileA')` is called. If we assume that thousands of files are being updated, the benefit of applying Delayed Timestamp Journaling can be substantial.

4.3 Logging while Swapping

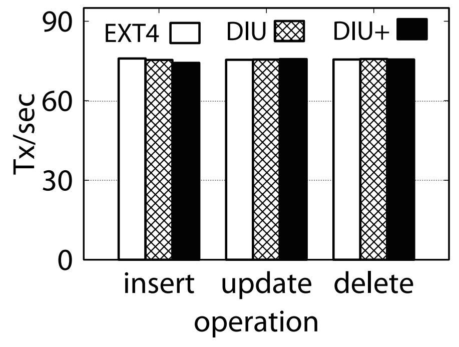
Linux kernel has a swapping mechanism which stores the less frequently used pages to the secondary storage. `kswapd` is responsible for swapping out the page cache entries [3]. In Delayed Inode Update scheme, the updated inode is inserted to a journal transaction when the dirty pages are flushed to the storage, i.e. `fsync()` and kworker thread. When flushing the dirty pages, the kworker thread identifies the dirty pages and inserts the metadata blocks associated with the dirty pages to the running transaction. If a dirty page is swapped out, the associated page cache entry is marked as clean. In Delayed Inode Update, the kworker thread fails to journal the updated inode if the associated dirty pages are all swapped out. In EXT4, this problem does not occur since the updated inode is inserted to the running transaction in the `write()` system call.

In Linux, only the page cache entries that have physical disk location are subject to swapping. The above-mentioned issue occurs only for the non-allocating writes.

In Delayed Inode Update, the swap daemon is modified to insert the updated metadata to the running transaction. When the swap daemon selects a victim page, it examines whether the associated inode block needs to be journaled. If necessary, the swap daemon inserts the associated metadata block to the running transaction.



(a) Random write



(b) SQLite

Figure 7: 4KB write followed by `fsync()`) and SQLite (persist mode). EXT4 runs in ordered mode.

5 EXPERIMENTS

We implement Delayed Inode Update in EXT4 of the Linux v4.10. We compare the behavior of three EXT4 versions: stock EXT4 (EXT4), EXT4 with Delayed Filesize Update (DIU) and EXT4 with Delayed Filesize Update and Delayed Timestamp Journaling (DIU+). In this experiment, we use PC server with Intel i5-2500, 8 GB RAM, Plextor M6 Pro 128 GB SSD, and Ubuntu 16.04 LTS.

5.1 Micro Benchmark

We use Mobibench [8] to generate workload. We use two workloads: random write and sqlite. In the random write workload, we create 256 MB file for each thread. A thread performs a 4KB random write followed by `fsync()`. We increase the number of threads from one to ten.

Fig. 7(a) illustrates the performances of EXT4, DIU and DIU+ in the random write workload (NAW-SP) respectively. The performance of DIU+ and DIU does not introduce any

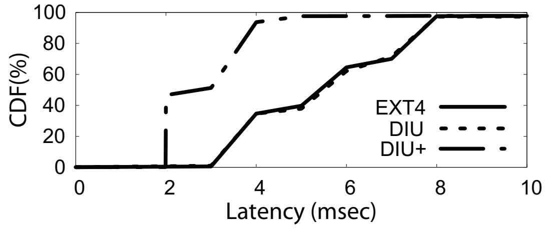


Figure 8: CDF of `fsync()` Latency in EXT4, DIU and DIU+

	mean	med	75th	99th	99.9th	99.99th
EXT4	20.6	6.78	8.07	379.3	4,439	4,877
DIU	21.2	6.80	8.10	383	4,494	4,500
DIU+	17.5	3.58	4.78	437.5	1,162	2,827

Table 2: `fsync()` latency statistics (msec)

performance degradation while they guarantee the atomicity in metadata update operation.

Fig. 7(b) shows the performance of SQLite transaction for EXT4, DIU and DIU+. Delayed Inode Update does not aggravate the IO performance while it guarantees the atomicity of the metadata update.

5.2 `fsync()` Performance

We examine the `fsync()` performance in EXT4, DIU and DIU+. The objective of this study is to examine the latency of the `fsync()`. We create 1,000 files and perform 4 KB sequential write on each of the files iteratively. All those files are the allocating writes. At the end of each iteration, we perform `fsync()` on the last written file. Each file is written total 4 MB. We measure the latency of `fsync()`.

Fig. 8 shows the result in CDF of the `fsync()` latency. Table 2 shows the mean, median, and the worst case latency. DIU+ reduced the median latency and the worst case latency of 99.99^{th} by 47% and 43% respectively, compared to EXT4. When `fsync()` is called, EXT4 writes the metadata of up to 1,000 files to the journal area at commit, including the inode related to the `fsync()`. However, DIU+ inserts only the inode associated with `fsync()` into the running transaction and writes the inode to the journal area. The EXT4 and DIU exhibit almost identical `fsync()` latency. That is because EXT4 and DIU share the same mechanism in inserting the updated metadata block to the running transaction and subsequently they exhibit the same amount of IO in filesystem journaling. On the other hand, the `fsync()` latency at 99.99% decreases by 40% by Delayed Timestamp Journaling from 4.9 sec to 2.8 sec. It reduces the latency of `fsync()` effectively when intensive `write()` and `fsync()` happens to a large number of files.

6 RELATED WORKS

Several papers have sought to improve the `fsync()` performance [5, 7, 9, 10, 12, 13, 17, 24]. IceFS [13] allocates a separate journal region called *Cube* for each container. It makes the journal commit operations for different containerized units more scalable and reduces the worst case latency of `fsync()` caused by a compound transaction [4, 15]. The filesystem journaling for a single cube suffers from the same worst case latency problem which the legacy EXT4 filesystem suffers from. Similar to IceFS, SpanFS [9] adopts independent virtualized storage devices, which is called a *domain* which is a logically separate unit for independent journaling among containers. By allocating a journal area for each domain, SpanFS can reduce the tail latency of the `fsync()` system call. However, SpanFS also has the `fsync()` latency problem caused by compound transactions in the same domain.

Eager syncing [6] suggests a logging mechanism to reduce the `fsync()` latency. For redo logging, Eager syncing constructs a set of data and metadata, which is called an *iset*. When `fsync()` is called, the iset associated with the system call is written to the logging area. By composing the iset only with the relevant data, Eager syncing reduces the latency for a `fsync()` system call. This differs from EXT4's compound transaction which puts unrelated metadata together; however, it has the overhead of journaling the data as well as the metadata related to the *iset*.

7 CONCLUSION

In this paper, we propose Delayed Inode Update scheme to address the atomic metadata update failure in EXT4. By delaying the update to the file size, we guarantee the atomicity of the metadata update operation. By delaying the time stamp journaling, we improve the tail latency of the `fsync()`. Through experiments, we show that Delayed Inode Update has negligible performance overhead and that it reduces the average and the worst case latency up to 15% and 43% respectively, compared to stock EXT4.

8 ACKNOWLEDGMENTS

We would like to thank Jan Kara for his feedback on our work. We would like to thank our shepherd, Abhigyan Sharma, for his insightful comment on this work. This work was in part supported by NRF (Basic Research Lab program, No. 2017R1A4A1015498) and by IITP (ICT & Future Planning program, No. R7117-16-0232).

REFERENCES

[1] Tizen Common Armv7 Images available. http://www.tizenexperts.com/2014/08/tizen-common-armv7l-images-available/.
[2] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving Ext4 for Shingled Disks. In *Proc. of USENIX Conference on File and Storage Technologies (FAST), 2017.*

[3] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proc. of IEEE Symposium on Security and Privacy (SP), 2007*.

[4] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *Proc. of ACM international systems and storage conference (SYSTOR), 2013*.

[5] Li-Pin Chang, Po-Han Sung, and Po-Hung Chen. Fast file synching for applications in flash-based android devices. In *Proc. of IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2014*.

[6] Li-Pin Chang, Po-Han Sung, Po-Tsang Chen, and Po-Hung Chen. Eager Synching: A Selective Logging Strategy for Fast fsync () on Flash-Based Android Devices. *ACM Transactions on Embedded Computing Systems (TECS), 2016*. 16, 2, 34.

[7] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices. In *Proc. of USENIX Conference on File and Storage Technologies (FAST), 2015*.

[8] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. AndroStep: Android Storage Performance Analysis Tool. In *Proc. of European Workshop on Mobile Engineering (ME), 2013*. 327–340. https://github.com/ESOS-Lab/Mobibench

[9] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *Proc. of USENIX Annual Technical Conference (ATC), 2015*.

[10] Yunji Kang and Dongkun Shin. Per-block-group journaling for improving fsync response time. In *Proc. of the 18th IEEE International Symposium on Consumer Electronics (ISCE), 2014*.

[11] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proc. of USENIX Conference on File and Storage Technologies (FAST), 2015*.

[12] Tae Hyung Lee, Minho Lee, and Young Ik Eom. An insightful write buffer scheme for improving SSD performance in home cloud server. In *Proc. of IEEE International Conference on Consumer Electronics (ICCE), 2017*.

[13] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Physical Disentanglement in a Container-Based File System. In *Proc. of Operating Systems Design and Implementation (OSDI), 2014*.

[14] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proc. of USENIX Annual Technical Conference (ATC), 2016*.

[15] Daejun Park, Min Ji Kim, and Dongkun Shin. Optimizing Fsync Performance with Dynamic Queue Depth Adaptation. *Journal of Semiconductor Technology and Science, 2015*, 15, 5, 571.

[16] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync (): A simple and efficient mechanism for preserving the integrity of durable data. In *Proc. of The European Conference on Computer Systems (EuroSys), 2013*.

[17] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proc. of USENIX Conference on File and Storage Technologies (FAST), 2017*.

[18] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014*.

[19] Vijayan Prabhakaran, Thomas L Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2008*.

[20] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS), 2013*, 9, 3, 9.

[21] Ohad Rodeh and Avi Teperman. zFS-a scalable distributed file system using object disks. In *Proc. of IEEE Conference on Mass Storage Systems and Technologies (MSST), 2003*.

[22] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating Performance and Energy in File System Server Workloads. In *Proc. of USENIX Conference on File and Storage Technologies (FAST), 2010*.

[23] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *Proc. of USENIX Conference on File and Storage Technologies (FAST), 2014*.

[24] Hankeun Son, Seongjin Lee, Gyeongyeol Choi, and Youjip Won. Coarse-grained mtime update for better fsync () performance. In *Proc. of ACM SIGAPP Symposium on Applied Computing (SAC), 2017*.

[25] Yongseok Son, Heon Yeom, and Hyuck Han. Optimizing I/O Operations in File Systems for Fast Storage Devices. *IEEE Transactions on Computers, 2016*.

[26] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System.. In *Proc. of USENIX Annual Technical Conference (ATC), 1996*.

[27] Theodore Ts'o. [PATCH] ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f3b59291a69d0b734be1fc8be489fef2dd846d3d.

[28] Stephen C Tweedie. Journaling the Linux ext2fs filesystem. In *Proc. of Annual Linux Expo, 1998*.

[29] Yupu Zhang, Chris Dragga, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. *-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In *Proc. of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2013*.